Making a chip (for devs)

1. Hello World Chip!

1.1 Building the Chip

1. Open the RecRoom repo in your preferred IDE.
2. Right click on the folder `Dependencies\com.recroom.circuitsv2\Runtime\Lang\Nodes` and create a new file called `TutorialNodes.cs`.
3. Paste in the following code:

```
using CircuitsV2.Lang;
using CircuitsV2.Lang.Nodes;
using CircuitsV2.Utilities;
using RecRoom.NoEngine.Common;
using System;
using System.Runtime.InteropServices;
using UnityEngine;

using static Circuits.RecRoom.Static.TypeUtil;
using static CircuitsV2.Factory.NodeUtil;

namespace CircuitsV2.Lang.Nodes
{
        [Guid("3b15431d-d461-403c-852a-97b774a80eac")]
        public sealed class HelloWorldNode : LocalNode
        {
                private static readonly NodeDesc Desc =
                        CNode("Hello World")
                                .CInput("", CExec)
                                .COutput("", CExec)
                        .End();

                public HelloWorldNode(CircuitGraph circuitGraph,
Guid nodeId)
                        : base(Desc, circuitGraph, nodeId) { }

                protected override Result<None> Run(ExecParams e,
ref InOut io)
                {
                        Debug.Log("Hello World!");
                        return Result.Ok();
                }
        }
}
```

4. Rebuild the solution in Unity (`Ctrl+r`).
5. Open the Circuits V2 window in Unity. It can be found in the application bar under 'RR Tools // CircuitsV2 // Window'.
6. Click the 'Generate Code and Assets' button.
   - Confirm changes to the following files.
     - Added:

        - 'Assets\Core\Scripts\CircuitsV2\ChipConfigs\CIRCUITSV2_GRAPH_NODES_HELLOWORLDNODE_ChipConfig.asset'
        - 'Assets\Core\Scripts\CircuitsV2\ChipConfigs\CIRCUITSV2_GRAPH_NODES_HELLOWORLDNODE_ChipConfig.asset.meta'
    - Modified:
        - 'Assets\Core\Scripts\CircuitsV2\CircuitsV2Config.asset'
        - 'Dependencies\com.recroom.circuitsv2\Runtime\CircuitNodeUtil.g.cs'
7. Rebuild the solution in Unity.

Your chip has been added to the game! Continue to see how to use it.

1.2 Using the Chip In-Game

1. Load into Rec Room in editor. If you have a room load override, disable it for now so that you load into your dorm.
2. Hold 1 to open your backpack and aim down to select your maker pen.
3. Press F to open up the tool menu.
    - We recommend setting the 'Snapping' setting to 'Surface' here. This makes chips snap to walls and align with each other.
4. Click the 'Open Palette' button on the tool menu.
5. Click the magnifying glass icon in the menu bar.
6. Click the search bar.
7. Type 'hello world'.
8. Click the 'Hello World' button.
9. Click on a wall to place your chip.
10. Click the search bar again.
11. Clear the search bar and type 'button v2'.
12. Click the 'Button V2' button.
13. Click to the right of your 'Hello World' chip to place a button.
14. Press F to open the tool menu.
15. Press the 'Wire' button to activate the wire tool.
16. Click and hold on the 'Pressed' port on the 'Button V2' control panel. While holding, drag to the input port on the 'Hello World' chip and release the mouse button. The chips should now be connected.
17. Click the part of the button that says 'Push Me!' to activate it. If you are still using the maker pen, you may need to click with the opposite mouse button. You should see the text "Hello World!" written to your log in Unity.

Fig 1.2.1 Hello World Graph

1.3 Discussion

Let's go through the code to create the chip line by line:

```
using CircuitsV2.Lang;
using CircuitsV2.Lang.Nodes;
using CircuitsV2.Utilities;
using RecRoom.NoEngine.Common;
using System;
using System.Runtime.InteropServices;
using UnityEngine;

using static Circuits.RecRoom.Static.TypeUtil;
using static CircuitsV2.Factory.NodeUtil;

namespace CircuitsV2.Lang.Nodes
{
    /** Hidden **/
}
```

The `using` statements bring in the classes required to implement a `Node`. Take note of the `using static` statements at the bottom. The `TypeUtil` class has static methods and properties for using our Circuits V2 types. By using `TypeUtil` statically we can use types like `CExec` without typing `TypeUtil.CExec`. We follow this pattern everywhere Circuits V2 types are used.

```
[Guid("3b15431d-d461-403c-852a-97b774a80eac")]
```

Every `Node` must have a unique guid. The guid in this `GuidAttribute` is used as its type id. `GuidAttribute` is included by `using System.Runtime.InteropServices;`. You can generate a guid in Visual Studio using the 'Create GUID' tool found in the application bar under 'Tools\Create Guid'. In the 'Create GUID' tool, you can click the 'New GUID' button to generate a guid and click the 'Copy' button to register it in your paste buffer. 'GUID Format' 5 is suitable for `GuidAttributes` on `Nodes`. Guids are used in many places in Circuits V2 aside from the `GuidAttribute` on `Node` classes. All guids in Circuits V2 must be universally unique. There are a few exceptions but are working to remove them.

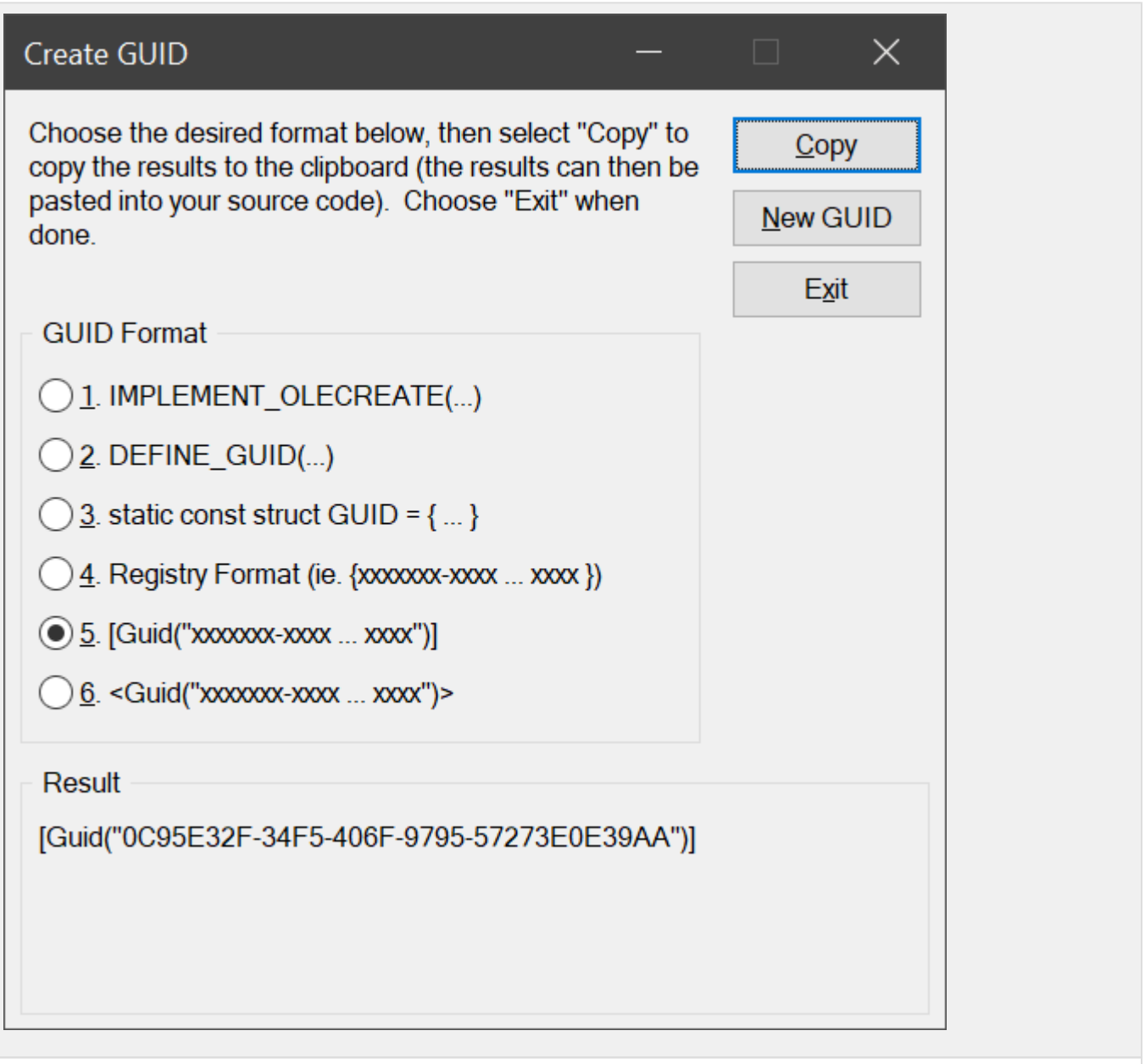
Create GUID
Choose the desired format below, then select "Copy" to copy the results to the clipboard (the results can then be pasted into your source code). Choose "Exit" when done.
Copy
New GUID
Exit
GUID Format
1. IMPLEMENT_OLECREATE(...)
2. DEFINE_GUID(...)
3. static const struct GUID = { ... }
4. Registry Format (ie. {xxxxxxx-xxxx ... xxxx })
5. [Guid("xxxxxxx-xxxx ... xxxx")]
6. <Guid("xxxxxxx-xxxx ... xxxx")>
Result
[Guid("0C95E32F-34F5-406F-9795-57273E0E39AA")]

Fig 1.3.1 Create GUID Window

```
public sealed class HelloWorldNode : LocalNode
{
  /** Hidden **/
}
```

Our `HelloWorldNode` inherits from `LocalNode`. `LocalNode` is one of the primary `Node` base classes. The full list is:

- `Node`: The most flexible base class. Clients should generally avoid extending `Node`. While it provides the most flexibility, it also requires the most customization. The base classes which extend `Node` tend to provide more fully-featured and optimized starting points and will keep client code consistent since they require changed lines to get up and running.
- `SimpleNode` (extends `Node`): The second most flexible base class. Initializes `NodeGroups` in a consistent way. `NodeGroups` are collections of `InputPorts` and `OutputPorts` that can be treated as a unit. A `NodeGroup` is analogous to a function in textual programming. `SimpleNodes` are optimized to work with one `NodeGroup` but they can have multiple if `IsDynamicDesc` is used. `Nodes` with multiple `NodeGroups` should generally be avoided.
- `LocalNode` (extends `SimpleNode`): The best starting place to implement a `Node` with a single `NodeGroup` inside of the Circuits V2 project (henceforth, Circuits V2). These `Nodes` can directly calculate results in their `Run()` method.
- `InteropNode` (extends `LocalNode`): The best starting place to implement a `Node` with a single `NodeGroup` outside of the Circuits V2. These nodes call into an `InteropDelegate` to calculate results. `InteropDelegates` are typically provided by the (henceforth, Rec Room).
- `OverrideNode` (extends `InteropNode`): Allows a `LocalNode` or `InteropNode` to be implemented which changes functionality depending on the types of its `InputPorts` and `OutputPorts`. An `OverrideNode` is similar to an overloaded function in textual programming.

We will talk about the different kinds of `Node` in more detail in later sections.

```
private static readonly NodeDesc Desc =
    CNode("Hello World")
        .CInput("", CExec)
        .COutput("", CExec)
    .End();
```

The `Desc` property is a `NodeDesc` which describes the ports on the `Node`. Our `Node` has 1 `CExec` input port and 1 `CExec` output port. The `CExec` type represents a sequence of execution starting at an `Event Receiver` node. Additional ports can be added with additional applications of the `CInput()` and `COutput()` functions like:

```
CNode("Hello World")
    .CInput("Input 1", CExec)
    .CInput("Input 2", CBool)
    .COutput("Output 1", CExec)
    .COutput("Output 2", CBool)
.End();
```

Notice that the first parameter of each function is a name, these appear next to the ports when the `Node` is spawned in Rec Room. The second parameter of each function is the type. Our second input and second output ports are both `CBool` type. This means they hold `true`/`false` values just like in other programming languages.

```
public HelloWorldNode(CircuitGraph circuitGraph, Guid nodeId)
    : base(Desc, circuitGraph, nodeId) { }
```

The constructor passes our `Desc` property down through to the `SimpleNode` base class. `SimpleNode` caches off the `Desc` and uses it to create a `NodeGroup` when `Initialize()` is called or to deserialize an existing `NodeGroup` in `Deserialize()`. `circuitGraph` is the graph the `Node` belongs to. It is eventually stored in `Node.CircuitGraph` and can be accessed througout the lifetime of the `Node`. `nodeId` is the `Node`'s unique id and is avaliable in the `Node.Id` property.

```
protected override Result<None> Run(ExecParams e, ref InOut io)
{
    Debug.Log("Hello World!");
    return Result.Ok();
}
```

The run method executes when the sequence of execution enters a port of `CExec` type. In the example, we created a button whose `Pressed` event began a sequence of execution passing through our 'Hello World' chip. Since our `Run()` method. Our `Run()` method logs "Hello World" to the screen.

2. Execution Nodes, Pure Nodes, and Parameters

2.1 Execution Nodes

The 'Hello World' chip we made earlier is an execution `Node`. An execution `Node` has at least one execution port. It usually has one input and one output but it can have zero, one or many of each. An execution `Node` runs `Execute()` when the sequence of execution enters one of its input `CExec` ports. Event receivers (such as buttons) are a special case which originate sequences of execution yet have no input `CExec` ports. C# code outside of Circuits V2 (usually in Rec Room) interacts with these `Nodes` to begin their execution.

2.2 Pure Nodes

Pure `Nodes` don't need execution ports, instead they run when one of their outputs is requested. Pure `Nodes` represent pure functions. A pure function always produces the same output given the same input. For example, `f(x) = 3 * x` has the same output for any given `x` no matter how many times you use that `x` with the function. The 'Hello World' chip is *not* pure because it modifies the state of the console window directly. In other words, the first time you run the 'Hello World' chip your console will say "Hello World" but if you run it again your console will say "Hello World Hello World". The most common reason a `Node` is not pure is because it is writing memory. In the case of the 'Hello World' chip, we are writing to the memory of the console window. You should *always* prefer to make `Nodes` pure if possible because they can be optimized more than execution `Nodes`.

Let's make a pure `Node` now. This code can be placed in TutorialNodes.cs:

```
// In usings
using System.Collections.Generic;

[Guid("12576e0c-15a7-4843-a23f-96c6c452f663")]
public sealed class TripleNode : LocalNode
{
    private static readonly NodeDesc Desc =
        CNode("Triple")
            .CInput("Value", CInt)
            .COutput("Result", CInt)
        .End();

    public InputPort Value => Input(0);
    public OutputPort Result => Output(0);

    protected override Result<None> In(ExecParams e) => InOut.In1(e,
this, Value);
```

```
    protected override IEnumerable<CircuitDelegate> Delegates =>
DelegateArray(
        (Result, ExprDelegate0)
    );

    public TripleNode(CircuitGraph circuitGraph, Guid nodeId)
        : base(Desc, circuitGraph, nodeId) { }

    protected override Result<None> Run(ExecParams e, ref InOut io) =>
        io.Return1(CircuitSignal.FromInt(io.in0.GetInt() * 3));
}
```

Build the Node in Unity using the instructions in section 1.1. To run this code, open Rec Room and spawn a 'Button V2', 'Triple' chip, 'To String' chip and 'Debug Log' chip. Connect the chips as shown in this image:

Fig 2.2.1 Triple Graph

Click on the 'Value' port on the 'Triple' chip with the wire tool active. This will allow you to configure its value. Set the value to 3 and press the 'Push Me!' button. You should see the text "9" written to your log in Unity.

2.3 Parameterized Execution Nodes

Lets see what changes when we add exec ports.

```
// In usings
using System.Collections.Generic;

[Guid("5b7ddfdd-dd09-455a-984a-5b8eee02f664")]
public sealed class TripleExecNode : LocalNode
{
    private static readonly NodeDesc Desc =
```

```
            CNode("Triple Exec")
                .CInput("", CExec)
                .CInput("Value", CInt)
                .COutput("", CExec)
                .COutput("Result", CInt)
            .End();

        public InputPort Value => Input(1);
        public OutputPort Result => Output(1);

        protected override Result<None> In(ExecParams e) => InOut.In1(e,
    this, Value);

        protected override IEnumerable<CircuitDelegate> Delegates =>
    DelegateArray(
            (Result, ExecDelegate0)
        );

        public TripleNode(CircuitGraph circuitGraph, Guid nodeId)
            : base(Desc, circuitGraph, nodeId) { }

        protected override Result<None> Run(ExecParams e, ref InOut io) =>
            io.Return1(CircuitSignal.FromInt(io.in0.GetInt() * 3));
    }
```

Use the chip in-game and notice that it's roughly the same as the 'Triple' chip except for the additional port.

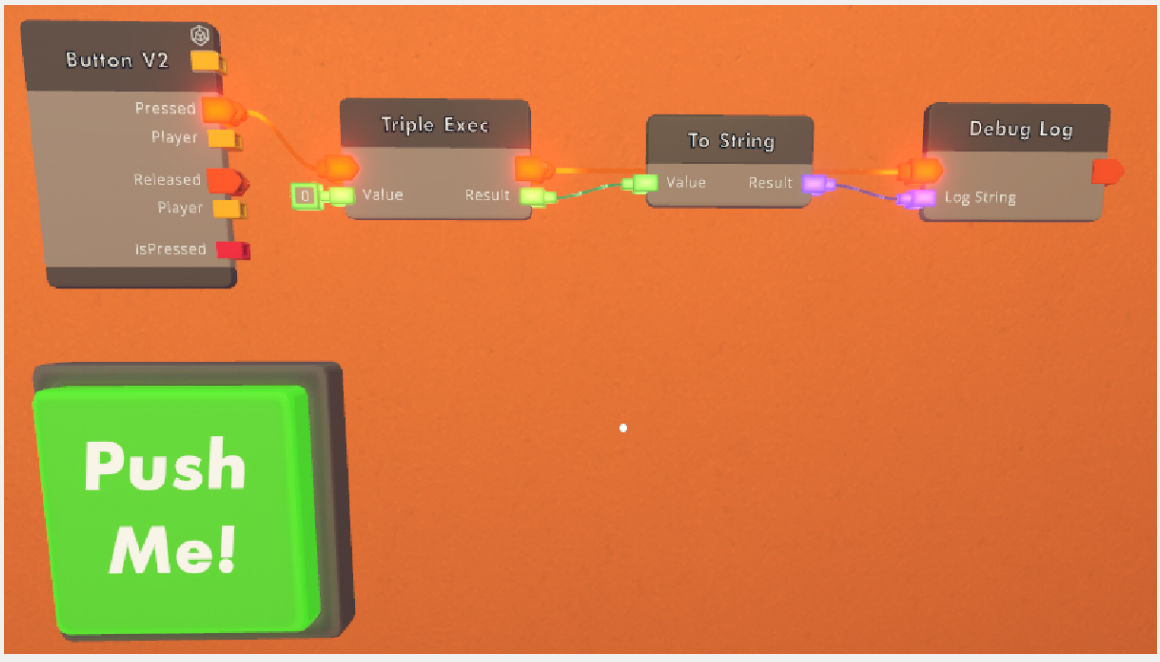

Fig 2.2.1 Triple Exec Graph

2.4 Discussion

We added some new lines of code these `Nodes`. To support their inputs and outputs. In the definition for `TripleNode` we see:

```
private static readonly NodeDesc Desc =
    CNode("Triple")
        .CInput("Value", CInt)
        .COutput("Result", CInt)
    .End();
```

These lines of code define a single `CInt` input and single `CInt` output for our node. This is the code that causes the node to spawn and render with ports in-game.

```
public InputPort Value => Input(0);
public OutputPort Result => Output(0);
```

The immediately following lines make it easy to access the ports by name. The `Input()` and `Output()` functions get ports at certain indices. By making these properties we can give them an easy-to-use name. Its worth noting here that the `TripleExecNode` defines these slightly differently.

```
public InputPort Value => Input(1);
public OutputPort Result => Output(1);
```

For `TripleExecNode` we use index `1` instead of `0`. Since this `Node` has `CExec` ports, we need to skip them.

```
protected override Result<None> In(ExecParams e) => InOut.In1(e, this,
Value);
```

This `In` function is called right before a `Node` runs. The `InOut` struct stores input and output values for a `Node`. It has `In()` functions numbered 1-8 which allow reading 1-8 `InputPorts` and storing their values. It also has corresponding `Return()` functions which store values that will be accessed by `OutputPorts`.

```
protected override IEnumerable<CircuitDelegate> Delegates =>
DelegateArray(
    (Result, ExprDelegate0)
);
```

The `Delegates` property provides a list of bindings from `OutputPort` to callback. The callback is executed when the `OutputPort` is read. The `(` in `ExprDelegate0` refers to `InOut.out0`. So `InOut.out0` will be returned when `Result` port is read. If this were `ExprDelegate1` it would refer to `InOut.out1` instead. This is another place where `TripleExecNode` differs from `TripleNode`.

```
protected override IEnumerable<CircuitDelegate> Delegates =>
DelegateArray(
    (Result, ExecDelegate0)
);
```

`TripleExecNode` uses `ExecDelegate0` instead of `ExprDelegate0`. Both `ExecDelegates` and `ExprDelegates` return the value of the corresponding data in the `InOut` struct. The difference is that `ExprDelegates` cause a `Node` to execute its `Run()` when an output port is read. `ExecDelegates` never execute their `Run()` functions from port reads, instead `Run()` executes when a sequence of execution enters a `CExec` port. All exec nodes should use `ExecDelegates` and all pure nodes should use `ExprDelegates`. At face value this detail makes pure nodes less efficient than their execution counterparts: requesting the same value multiple times may start an entire chain of execution. Fortunately, we can make an optimization: we cache the values of all `OutputPorts` for pure nodes between calls to execution nodes. In other words, if an execution node runs and then a bunch of pure nodes run right after, those pure nodes won't be recalculated again until the next execution node runs.

The rest of the code is similar to our original `HelloWorldNode`.